```java
import java.util.Iterator;

public class SimpleLinkedList<ELEMENT extends Comparable<ELEMENT>> implements
ILinkedList<ELEMENT> {

    //region Node Class

    private class Node<ELEMENT> {
      public ELEMENT item;
      public Node<ELEMENT> next;

      public Node() {
        this(null, null);
      }
      public Node(ELEMENT item) {
        this(item, null);
      }
      public Node(ELEMENT item, Node<ELEMENT> next) {
        this.item = item;
        this.next = next;
      }

      @Override
      public String toString() {
        return this.item.toString();
      }
    }
    //endregion

    //region Attributes

    protected Node<ELEMENT> head;
    protected int count;
    protected Node<ELEMENT> tail;
    //endregion

    //region Constructors

    public SimpleLinkedList() {
      this.head = null;
      this.count = 0;
      this.tail = null;
    }
    //endregion

    //region Linked List Methods

    // Returns the number of elements in this list.
```

```
public int size() {
  return this.count;
}

public void addFirstRookieVersion(ELEMENT item) {
  if (this.count == 0) {
    this.head = this.tail = new Node<ELEMENT>(item, null);
    ++this.count;
  } else {
    Node<ELEMENT> temp = new Node<ELEMENT>(item, null);
    temp.next = this.head;
    this.head = temp;
    ++this.count;
  }
}
// Inserts the specified element at the beginning of this list.
public void addFirst(ELEMENT item) {
  Node<ELEMENT> temp = new Node<ELEMENT>(item, this.head);
  if (this.count == 0) {
    this.tail = temp;
  }
  this.head = temp;
  ++this.count;
}

public void addLastRookieVersion(ELEMENT item) {
  if (this.count == 0) {
    this.head = this.tail = new Node<ELEMENT>(item, null);
    ++this.count;
  } else {
    Node<ELEMENT> temp = new Node<ELEMENT>(item, null);
    this.tail.next = temp;
    this.tail = temp;
    ++this.count;
  }
}
// Appends the specified element to the end of this list.
public void addLast(ELEMENT item) {
  Node<ELEMENT> temp = new Node<ELEMENT>(item, null);
  if (this.count == 0) {
    this.head = temp;
  } else {
    this.tail.next = temp;
  }
  this.tail = temp;
  ++this.count;
}

// Removes and returns the first element from this list.
```

```java
    public ELEMENT removeFirst() {
        if (this.count == 0) {
            throw new RuntimeException("La lista está vacía...");
        }
        ELEMENT item = this.head.item;
        this.head = this.head.next;
        if (this.head == null) {
            this.tail = null;
        }
        --this.count;
        return item;
    }

    // Removes and returns the last element from this list.
    public ELEMENT removeLast() {
        if (this.count == 0) {
            throw new RuntimeException("La lista está vacía...");
        }
        ELEMENT item = this.tail.item;
        if (this.head.next == null) {
            this.head = this.tail = null;
        } else {
            Node<ELEMENT> skip = this.head;
            while (skip.next.next != null) {
                skip = skip.next;
            }
            this.tail = skip;
            this.tail.next = null;
        }
        --this.count;
        return item;
    }
    //endregion

    //region Object Methods

    @Override
    public String toString() {

        if (this.size() <=0) {
            return "";
        }

        // from https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/
StringBuilder.html
        StringBuilder sb = new StringBuilder();

        sb.append("[" + this.head.item.toString());
        for (Node<ELEMENT> skip = this.head.next; skip != null; skip = skip.next) {
```

```
            sb.append(", " + skip.item.toString());
        }
        sb.append("]");

        return sb.toString();
    }
    //endregion



    //region Iterable Methods
    @Override
    public Iterator<ELEMENT> iterator() {
        return new SimpleLinkedListIterator(this.head);
    }

    private class SimpleLinkedListIterator implements Iterator<ELEMENT> {
        private Node<ELEMENT> current;

        public SimpleLinkedListIterator(Node<ELEMENT> current) {
            this.current = current;
        }

        @Override
        public boolean hasNext() {
            return this.current != null;
        }

        @Override
        public ELEMENT next() {
            if (!this.hasNext()) {
                throw new RuntimeException("La lista está vacía...");
            }
            ELEMENT item = this.current.item;
            this.current = this.current.next;
            return item;
        }

    }

    // AGREGAR ELEMENTO EN UNA POSICIÓN DADA

    public void InsertElement(ELEMENT item, int pos) {
        //if (pos < 0 || pos >= this.count) { // Se verifica que la posición este dentro del intervalo
        //    throw new IndexOutOfBoundsException("Posición no valida (" + pos + ")"); // Se arroja
un error
        //}
        if (pos == 0) { // Si la posición es 1
            addFirst(item); // Se agrega al principio
```

```
        } else if (pos == this.count) { // en caso, que la posición sea igual a la cantidad de elementos
            addLast(item); // Se agrega al final
        } else {
            Node<ELEMENT> skip = this.head; // Head apunta al primer nodo, este valor se lo pasa a skip (skip es el puntero que voy a usar para recorrer la lista)
            for (int i = 0; i < pos - 1; i++) { // Se avanza hasta la posición anterior al nodo a agregar
                skip = skip.next; // next permite que el puntero se siga moviendo hasta el siguiente nodo, ahora skip esta apuntando al nodo anterior a la posición a insertar
            }

            Node<ELEMENT> temp = new Node<>(item, null); // Se crea el nodo temp -> este contiene el valor de item, pero no apunta a ningún lado
            temp.next = skip.next; // Conectar el nuevo nodo con el siguiente -> Si temp.next [x] ahora apunta a [c] ([a][b][x][c]) conecta [X] con el siguiente ([C]) [A]→[B]→[C], ahora [X]→[C]
            skip.next = temp; // Conectar el nodo anterior con el nuevo  -> [b] -> [x] conecta el anterior ([B]) con el nuevo ([X]) [A]→[B]→[X]→[C]
            ++this.count; // Aumentar el tamaño de los nodos
        }
    }

    // ELIMINAR ELEMENTO EN UNA POSICIÓN DADA

    public void DeleteElement(int pos){
        //if (pos < 0 || pos >= this.count){
        //    throw new IndexOutOfBoundsException("Posición no valida (" + pos + ")");
        //}
        if (pos == 0){
            removeFirst();
        }else if (pos == this.count - 1){
            removeLast();
        }else{
            Node <ELEMENT> skip = this.head;
            for (int i = 0; i < pos - 1; i++){
                skip = skip.next;
            }
            ELEMENT item = skip.next.item; // Opcional, guardar el valor del nodo a borrar
            skip.next = skip.next.next; // conectar el nodo anterior (al que se va a eliminar) con el siguiente (del que se eliminó).
            --this.count; // Disminuyo
        }
    }

    // AGREGAR ELEMENTOS EN ORDEN -> ASCENDENTE

    public void addInOrder(ELEMENT item){
        if (this.head == null){ // Si la lista no está vacía, cargo el primer elemento como head
            addFirst(item); // Cargo el primero
        }else if (item.compareTo(this.head.item) <= 0){ // -1 o 0 significa que es menor o igual a head
            addFirst(item); // Por lo tanto agrego al principio
```

```
        } else if (item.compareTo(this.tail.item) > 0){ // 1 indica que el elemento es mayor a tail
            addLast(item); // Por lo tanto se agrega al final
        }else{ // Caso contrario -> se agrega en el medio
            Node <ELEMENT> skip = this.head; // Creo el puntero skip
            Node <ELEMENT> prev = null; // Guardamos el nodo anterior que estamos analizando
            while (skip != null && item.compareTo(skip.item) > 0){ // Mientras no terminamos de
recorrer la lista, y el item a cargar es mayor al actual
                prev = skip; // Se guarda el nodo actual -> al finalizar apunta al nodo mayor a item
                skip = skip.next; // Se avanza al siguiente nodo -> apunta  al nodo anterior a ese
            }
            Node<ELEMENT> temp = new Node<>(item, skip); // Se crea el nuevo nodo que contiene el
valor de item
            // Conectar prev con el nuevo nodo (prev nunca debería ser null aquí, pero chequeamos
por seguridad)
            if (prev != null) { // Si prev es distinto de null
                prev.next = temp; // Se une temp, el nuevo nodo, con el enlace anterior
            } else { // Caso contrario
                addFirst(item); // Se agrega al principio
            }
        }
    }

    //endregion

}
```